

## Algoritmos y Estructuras de Datos II

### Segundo parcial — 1<sup>er</sup> cuatrimestre 2018

- El parcial es a libro abierto.
- Cada ejercicio debe entregarse en hojas separadas.
- Incluir en cada hoja el número de orden asignado, número de hoja, LU, apellido y nombre.
- Antes de entregar, **remover los "pelitos" del borde de las hojas**, si hubiere.
- Cada ejercicio se calificará con Perfecto, Aprobado, Regular, o Insuficiente.
- El parcial está aprobado si el ejercicio 1 tiene al menos A, y entre los ejercicios 2 y 3 hay al menos una A.

P	A	A	Aprobado
①	②	③	Nota

### Ej. 1. Diseño

Un sistema operativo (SO) ejecuta muchos programas en simultáneo y mantiene un conjunto de recursos dedicados que los programas utilizan. En cualquier momento pueden iniciarse programas. Además, en cualquier momento un programa  $p$  puede solicitar el uso de un recurso  $r$  del sistema, siempre que  $p$  no esté utilizando otro recurso. Si  $r$  está libre, el SO se lo asigna a  $p$  en forma dedicada y  $p$  continúa. Si  $r$  estaba asignado a algún otro proceso, el SO bloquea a  $p$  y lo deja a la espera de que  $r$  se libere. Cuando un proceso termina de utilizar un recurso  $r$ , lo informa al SO y éste asigna automáticamente  $r$  a otro proceso que esté bloqueado esperando por  $r$  (de haberlo). Si hubiese más de un proceso bloqueado esperando por  $r$ , el SO respeta el orden cronológico de las solicitudes.

#### TAD SO

##### observadores básicos

procesos	: so	→ conj(proceso)	
recursos	: so	→ conj(recursos)	
enUso	: so $s \times$ proceso $p \times$ recurso $r$	→ boolean	$\{p \in \text{procesos}(s) \wedge r \in \text{recursos}(s)\}$
esperando	: so $s \times$ recurso $r$	→ cola(proceso)	$\{r \in \text{recursos}(s)\}$

##### generadores

iniciar	: conj(recurso)	→ so	
lanzarProceso	: so $s \times$ proceso $p$	→ so	$\{p \notin \text{procesos}(s)\}$
solicitar	: so $s \times$ proceso $p \times$ recurso $r$	→ so	$\{p \in \text{procesos}(s) \wedge r \in \text{recursos}(s) \wedge p \notin \text{bloqueados}(s) \wedge \neg \text{usandoRecurso}(s,p)\}$
liberar	: so $s \times$ proceso $p$	→ so	$\{p \in \text{procesos}(s) \wedge p \notin \text{bloqueados}(s)\}$

##### otras operaciones

bloqueados	: so $s$	→ conj(proceso)	
siendoUsados	: so $s$	→ conj(recurso)	
usandoRecurso	: so $s \times$ proceso $p$	→ boolean	$\{p \in \text{procesos}(s)\}$

**axiomas**  $\forall s: \text{so}, \forall p, p': \text{proceso}, \forall r, r': \text{recurso}$

...

Fin TAD

Suponiendo que tanto los procesos como los recursos se representan con números naturales, se debe realizar un diseño que cumpla con los siguientes órdenes de complejidad en el peor caso, siendo  $R$  la cantidad de recursos del sistema:

- lanzarProceso( $p$ ):  $O(\log p)$
- solicitar( $p, r$ ):  $O(\log p + \log R)$
- liberar( $p$ ):  $O(\log p)$
- bloqueados() y siendoUsados(): ambas en  $O(1)$

**Observación:** notar que las complejidades dependen del proceso  $p$  y no de la cantidad de procesos del SO. Aun liberar un recurso, sólo depende del proceso que lo libera, no del otro proceso que toma el recurso liberado.

1. Escriba la estructura de representación del módulo SO explicando detalladamente qué información se guarda en cada parte de la misma y las relaciones entre las partes. Describa también las estructuras de datos subyacentes.
2. Escriba los algoritmos para solicitar y para liberar un recurso utilizado por un programa y justifique el cumplimiento de los órdenes solicitados. Para las demás funciones, descríbalas en castellano, justificando por qué se cumple el orden de complejidad pedido.



## Ej. 2. Ordenamiento

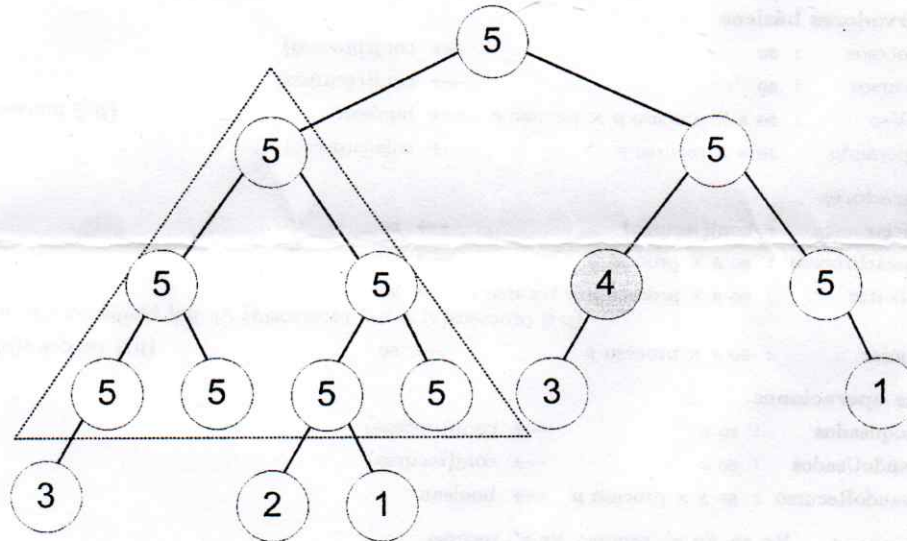
En un campeonato de fútbol, un partido se registra como una tupla  $\langle e_1: \text{string}, e_2: \text{string}, g_1: \text{nat}, g_2: \text{nat} \rangle$ , donde  $e_1$  y  $e_2$  son los equipos que se enfrentaron (strings no vacíos) y  $g_1$  y  $g_2$  son los goles de cada uno en ese partido, respectivamente. Por cada partido, un equipo recibe 3 puntos si ganó, 1 punto si empató y 0 puntos si perdió.

Se pide escribir un algoritmo que tome un arreglo con los partidos del campeonato y arme la *tabla de posiciones* con todos los equipos del campeonato (i.e., los que figuran en algún partido del arreglo). La tabla de posiciones es una lista con los equipos ordenada decrecientemente según los puntos obtenidos. En caso de empate de puntos, se desempata con la cantidad de goles a favor de cada equipo. Puede haber partidos que se jueguen más de una vez (potencialmente con diferentes resultados).

Siendo  $[p_1, p_2, \dots, p_n]$  el arreglo de entrada con  $n$  partidos y sabiendo que la cantidad de goles hechos por cada equipo en cada partido está acotada por una constante  $k$ , se espera que el algoritmo propuesto tenga una complejidad lineal en función del tamaño de entrada, es decir,  $O(S)$ , donde  $S = \sum_{i=1}^n (|p_i.e_1| + |p_i.e_2|)$ .

## Ej. 3. Dividir y Conquistar

La *copa* de un árbol binario de naturales está formada por los primeros niveles (completos) del árbol cuyos elementos coincidan con el de la raíz. Por ejemplo, el siguiente árbol tiene una copa de tamaño 2 (formada por el nodo raíz y sus dos hijos), aunque si el nodo marcado en gris tuviese un 5 en lugar de un 4, el árbol tendría una copa de tamaño 3.



Se pide diseñar un algoritmo de tipo *Divide & Conquer* que dado un árbol binario de naturales, indique el tamaño máximo de copa entre todos los subárboles del mismo (consideramos que un árbol es también subárbol de sí mismo). En el ejemplo anterior, la respuesta sería 3, dada por el subárbol izquierdo de la raíz del árbol (cuya copa de tamaño 3 está enmarcada en el triángulo).

El algoritmo diseñado debe tener una complejidad de peor caso de  $O(n)$ , siendo  $n$  la cantidad de nodos del árbol. Se pide además:

1. Justificar la complejidad del algoritmo suponiendo que el árbol está balanceado.
2. Justificar la complejidad del algoritmo en el caso en que el árbol no esté balanceado.



(1) Se se representa con err  
 donde err es tupla  $\langle$  ~~procesos~~: dictTrie(proceso, itDicAVL),  
Recursos: dictAVL(recurso, tupla  $\langle$  cola(itDicTrie), itC: itConflines,  
 itD: itDicTrie  $\rangle$ ), bloqueados: confTrie(procesos),  
siendoUsados: conflines1(recursos)  $\rangle$

(±) En procesos guardo todos los procesos del SO y  
 en el significado un iterador ~~de recursos~~ ~~de~~  
~~este iterador~~  $\rightarrow$  un recurso (el usado actualmente por  
 el proceso o el solicitado) en el diccionario recursos.  
 En este último, como significado, hay una tupla  
 con una cola de iteradores del diccionario procesos  
 (donde se encuen los procesos que solicitan el recurso),  
 y un iterador  $\rightarrow$  un recurso en el conf siendoUsados  
 donde se guardan los recursos que están siendo  
 utilizados <sup>y un iterador a proceso (indica el proceso actual si lo hay).</sup>  
 En el conjunto bloqueados se guardan  
 los procesos que ya solicitaron un recurso y están  
 a la espera.

Procesos es un diccionario sobre Trie, lo que permite  
 buscar, insertar y ~~eliminar~~ eliminar en  $O(\log(p))$  con  $\log(p)$ -  
 cantidad de dígitos de  $p$ , ya que el trie recorre  
 los dígitos de los elementos.

Recursos es un diccionario sobre AVL, lo cual permite

busca en  $O(\log(2))$  ya que es un árbol balanceado y se realiza una búsqueda binaria.

La ~~lib~~ `colb(dicTrie)` permite añadir y eliminar en  $O(1)$  ya que necesitamos solo insertar o no y remover o no.

El conjunto sobre Trie ~~de~~ ~~para~~ (bloques) sigue la misma lógica del ~~dic~~ diccionario sobre ~~trie~~ trie ya que permite insertar, eliminar y buscar en  $O(\log(d))$ .

`siendoUsado` es un conjunto lineal que permite insertar en  $O(1)$  y obtener un iterador al ~~primer~~ <sup>siguiente elemento iterado</sup> también en  $O(1)$ .

## (II) • Lanzar Proceso (P)

Para este ~~algoritmo~~ algoritmo lo único que hay que hacer es insertar el proceso P en procesos con significado en Hereder mto. Esto se logra en  $O(\log(d))$  ya que ~~buscamos~~ es una inserción en un Trie.

### • `bloques()`

Se devuelve la componente `bloques` de la tupla en  $O(1)$ . *Es importante decir que se devuelve por ref.*

### • `siendoUsado()`

Lo mismo que `bloques()` devolviendo la componente `siendoUsado`.



• Solicitar( $p, r, w, e$ ):

$itp \leftarrow \text{Buscar}(p, e.\text{procesos}) \quad // O(\log(p))$

$itr \leftarrow \text{Buscar}(r, e.\text{recursos}) \quad // O(\log(R))$

if  $\text{Activo}(itr).T_2.T_3 = itr.\text{color\_cabo}$  then

$\text{Activo}(itp).T_2.T_3 = itp$   $// O(1)$

insertar(~~itr~~, ~~e.siendo usados~~, e.siendo usados)  $// O(1)$

$\text{Activo}(itr).T_2.T_2 = itr$  color al último elemento de e.siendo usados  $// O(1)$

eliminar( $itp, \text{Activo}(itr).T_2.T_1$ )  $// O(1)$

Insertar( $p, e.\text{bloqueos}$ )  $// O(\log(p))$

endif

$\text{Activo}(itp).T_2 = itr$

Idea del algoritmo: si el recurso solicitado( $r$ ) ~~no~~ está siendo usado por un proceso, dado por la última componente de su significado, entonces cambio esa componente por  $p$ , inserto  $r$  en siendo usados y pongo el iterador de ese componente en la tercer componente de  $r$ . Si el recurso ~~no~~ está siendo usado elimino el proceso a la cabecera de  $r$  e inserto  $p$  a los bloqueos

$$\text{Complejidad} = O(\log(p)) + O(\log(r)) + O(1) =$$

$$O(\log(p) + \log(r)) \quad \text{Busco e inserto en un tree y busco en un tree.}$$

Al final como significado del proceso pongo el iterador al recurso solicitado



•  $Liberar(p, e)$ :

$itp \leftarrow Buscar(p, e \text{ procesos}) \quad // O(\log(p))$

$itr \leftarrow Actual(itp).T_2 \quad // O(1)$

if  $Actual(itr).T_1 = \text{cob vacío then}$   $// O(1)$

Eliminar( $Actual(itr).T_2.T_2$ )  $// O(1)$

$Actual(itr).T_2.T_2 = \text{iterador nulo}$   $// O(1)$

else

$Actual(itr).T_2.T_3 = \text{próximo}(Actual(itr).T_2.T_1) \quad // O(1)$

Eliminar( $\text{próximo}(Actual(itr).T_2.T_1)$ , e bloqueos)  $// O(\log(p))$

Desenrollar( $Actual(itr).T_2.T_1$ )  $// O(1)$

endif

$Actual(itp).T_2 = \text{iterador nulo}$   $// O(1)$

Idea del algoritmo: en base  $p$  en procesos y obengo su significado(r)

~~proceso bloqueado~~, si  $r$  tiene procesos en

cola ~~el~~ elimo a r de estado Utilizados y

pago el iterador ~~de~~ de proceso que estaba utilizando el recurso en nulo (tercer componente de la tupla).

Si r tiene procesos ~~bloqueados~~ en cola pago al próximo proceso ~~como~~ en la tercera componente del significado de r, elimino al próximo de e.bloqueados y desenrolla.

Al final pago como significado de p en e.procesos el iterador nulo.



$$\text{Complejidad} = O(\log(p)) + O(1) + O(\log(p)) = O(\log(p))$$

ya se realiza una búsqueda y una renovación en un trie con complejidad  $O(\log(p))$  c/v.

(Perdón esto es un anexo)

Estaba todo bastante prolijo, solo trataba de hacer los ejercicios en hojas separadas 😊

- ② Recorro el arreglo de tuplas e inserto en un diccionario sobre trie todos los equipos. Como significado ~~de cada equipo~~ va a tener una tupla de ~~un~~ natural, ~~real~~, una ~~para~~ los puntos y otra ~~por~~ la diferencia de goles. ~~(O(3))~~

Recorro de nuevo el arreglo y voy calculando para cada equipo los puntos (si  $g_1 > g_2$  sumo 3 a  $e_1$ , si  $g_1 = g_2$  sumo 1 a ambos y si  $g_1 < g_2$  sumo 3 a  $g_2$ ) y la diferencia de goles (sumo  $g_1 - g_2$  a  $e_1$  y  $g_2 - g_1$  a  $e_2$ ).  $(O(S))$

¿Está bien esto, pero por qué es  $O(S)$ ?

Voy recorriendo los elementos del trie y hago un bucket sort con la diferencia de goles (que está acotada)  $(O(m))$

No hace falta armar un AVL, puedes hacer un bucket igual al anterior pero con puntos y va a ser estable

Recorro el vector de buckets y los buckets y voy insertando ~~según tuplas~~ ~~(pero que son estables)~~ en

un diccionario ~~(no, string)~~ según los puntos. Insertando ~~arriba~~ en la lista para que sea estable. Después recorro en ~~orden~~ ~~inverso~~ el diccionario y paso a un vector en orden decreciente.  $(O(m))$

Como la complejidad es  $O(S+m) = O(S)$



(3)  $\text{MaxCops}(in A: \text{arb}(A)) \rightarrow \text{not}$   
 if  $\text{cont}(A) \neq 2$  then ~~return~~  $\text{return } 1$  (la raíz sola)

else

$\text{der} \leftarrow \text{MaxCops}(\text{der}(A))$  ~~return~~

$\text{izq} \leftarrow \text{MaxCops}(\text{izq}(A))$  ~~return~~

if  $\text{RAIZ}(A) = \text{RAIZ}(\text{der}(A)) \wedge \text{RAIZ}(A) = \text{RAIZ}(\text{izq}(A))$  then ~~return~~  $\text{return } 1$

if  $\text{der} \neq \text{izq}$  then

~~return  $\text{max}(\text{der}, \text{izq})$~~  ~~return el más chico entre los dos~~

else

~~return~~  $1 + \text{der}$

endif

else

~~return~~  $\text{max}(\text{der}, \text{izq})$

endif

endif

Perdón por el tachón, me confundí: P

(I) Si el árbol está balanceado hacer  $\text{MaxCops}(\text{der}(A))$   
 y  $\text{MaxCops}(\text{izq}(A))$  cuesta  $T(\frac{n}{2})$  clu y resto  
 el resto cuesta  $\Theta(1)$

Luego,  $T(n) = 2T(\frac{n}{2}) + \Theta(1)$

Usar el teorema maestro:

$a=2, b=2, f(n) = \Theta(1)$

Cae en el primer caso ya que  $f(n) = \Theta(1) \leq O(n^{\log_2(2)-\epsilon}) = O(n^{1-\epsilon})$   
 para algún  $\epsilon > 0$  cte, entonces



$$T(n) \leq \Theta(n^{\log_2(2)}) = \Theta(n) \quad \checkmark$$

(II) En el caso de que el árbol no esté balanceado no partimos el problema a partes de tamaño  $\frac{n}{2}$ , pero la suma de las partes va a ser  $n$  y estoy recorriéndolo una única vez cada nodo del árbol en peor caso. Por lo tanto la complejidad del algoritmo no se ve afectada. ~~OK~~